Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 207 (2008) 153–169

www.elsevier.com/locate/entcs

Engineering of An Assertion-based PSL^{Simple}-Verilog Dynamic Verifier by Alternating Automata

Naiyong Jin, Chengjie Shen, Jun Chen and Taoyong Ni[1,2]

Software Engineering Institute
East China Normal University
Shanghai, China

Abstract

Alternating Finite Automata (AFA) has linear space complexity in representing Linear-Time Temporal Logics. However, It is difficult to manipulate AFA in the run-time. In this paper, we focus on implementation methods to make alternating automata from static representation to run-time verification engines. 1) We have Directed Acyclic Graphs (DAG) represent all possible runs of a Local-variable-enhanced AFA (LAFA). The acceptance of universal choices is conditioned on successful synchronization of universal branches. 2)We encode states and local variables by symbolic approaches, and adopt historic trees in representing all possible parallel runs. The encoding enables multiple assignments to states and local variables in a configuration. By those methods, we are able to maintain the linear complexity of verification in both space and time.

Keywords: Assertion-based Verification, Automata Construction, Property Specification Language

1 Introduction

Assertion-based dynamic verifiers automatically pick up execution traces, which satisfy or violate certain property assertions, during the simulation of DUVs (Design Under Verification). The automation can significantly reduce the verification cost and promote the design quality [12]. Therefore, assertion-based verification is becoming a more and more important engineering practice.

PSL [3] is an industry standard specification language (IEEE-1850) for hardware and embedded system design. PSL has many features supporting simulation, including the directives for the assertions, the test range restrictions, and the functional coverages. The simple subset of *PSL* (PSL^{Simple}) conforms to the notion of

[1] This paper is supported by the "Dengshan Project"(067062017) of the Science and Technology Commission of Shanghai Municipality.

[2] Email:{nyjin, cjshen, jchen, ytni}@sei.ecnu.edu.cn

doi:10.1016/j.entcs.2008.03.091

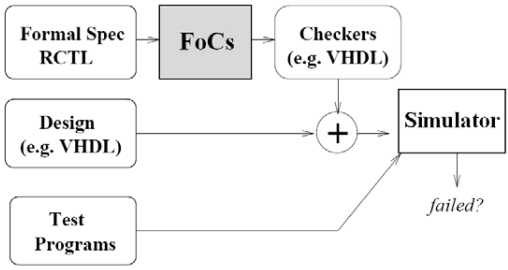


Fig. 1. FoCs Environment

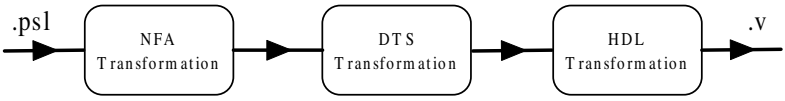


Fig. 2. A *PSL* Formula Transformation Flow for Dynamic Verification

monotonic advancement of time, which in turn ensures that formulas within the subset can be simulated easily. Verilog-HDL (IEEE-1364) [1] is an implementation language popularly used in circuit design.

In this paper, we introduce a PSL^{Simple}-Verilog dynamic verifer and discuss methods for its implementation.

Related Work

In [2], Abarbanel *et. al.* presented the framework FoCs (Formal Checkers, Fig. 1) for generating property checkers from RCTL specifications. The checkers are integrated into DUVs. During simulation, the checkers monitor the execution of a design and identify violations of the property assertions. Following Abarbanel's approach, Pidan *et al.* [20] proposed an optimized algorithm for dynamic verifiers of PSL formulas. Firstly, they transformed a PSL formula to a Non-deterministic Finite Automaton(NFA). Then they implemented the NFA with a Discrete Transition System(DTS), which, in turn, will be translated into Verilog HDL codes. The transformation flow is illustrated in Fig. 2

Fisman *et al.* [4] defined the core logic of PSL as LTL_WR, an extension of LTL with regular expressions. They proved that for every LTL_WR formula f there exists a Non-deterministic Büchi Automaton (NBA) whose size is *doubly exponential* in the size of f. As a NFA is the finite fragment of a NBA, the algorithm of Pidan has the same complexity as that of Fisman. The exponential increase of states comes from the intersection (length-matching conjunction) of regular expressions $r_1 \&\& r_2$. Fisman [10] also defined a subset of *PSL* formulas, whose violations can be detected by linear automata on finite words. The subset did not include $r_1 \&\& r_2$.

Alternating Finite Automata (AFA) [4] are exponentially more succinct than NFA in expressing temporal logic formulas. The size of resulting AFA is linear to that of f. Feikbeiner and Sipma [9] proposed three algorithms to check at run-time whether a reactive system satisfies a LTL specification by AFA. Those algorithms traversed an AFA in different ways: breadth-first search, width-first search and

backward search. Feikbeiner's methods reflect the exponential time-complexity in handling AFA.

Fisman *et al.* also used AFA as an intermediate form in transforming LTL_WR to NFA. One important reason that forced them to further transform AFA to NFA was due to the fact that

> *Alternation in general may lead to automata runs in which each branch is accepting, while at the same time, the simultaneously visited states may include accepting and some non-accepting states at all times* (in Section 4 of [4])

The statement conveys the idea that the traditional AFA do not have accepting states for the length-matching construction $r_1\&\&r_2$. As a consequence, it is impossible to *sequentially concatenate* and *fuse* the AFA of $r_1\&\&r_2$ with others.

However, it is always enticing to use AFA as verification engines [14] [21]. In [16], we solved Fisman's problem by enhancing AFA with *local variables* (LAFA) so that our automata constructions had accepting states for all PSL^{Simple}'s regular expressions, and were able to distinguish different satisfaction strengths. Here, we focus on implementation methods to turn the LAFAs from representation to dynamic verification engines.

Contribution

In summary, we contribute to the literature in the following aspects.

(i) We have Directed Acyclic Graphs (DAG) represent all possible runs of a LAFA. The acceptance of universal choices is conditioned on the successful synchronization of universal branches.

(ii) We encode states and local variables by symbolic approaches, and adopt historic trees in representing all possible parallel runs.

By those methods, we managed to avoid breadth-ward searching. Consequently, the complexity of our algorithm is linear in both space and time.

2 PSL^{Simple}: A Subset of PSL for Dynamic Verification

PSL has four layers of language structures: Boolean, temporal, verification and modelling. The temporal layer is the heart of *PSL*. It is used to describe complex temporal relations between signals. The temporal layer of *PSL* supports regular expressions, linear temporal logic and branching temporal logic. Here, we work on formulas in regular expressions and linear temporal logic.

The simple subset of PSL (PSL^{Simple}) is a subset that conforms to the notion of monotonic advancement of time, left to right through the property, which in turn ensures that properties within the subset can be simulated easily.

PSL^{Simple} restricts operand types of temporal formulas. Let b range over Boolean expressions, r range over Sequential Extended Regular Expressions(SEREs), and f range over PSL^{Simple} formulas.

The set of SEREs is defined recursively as follows:

Definition 2.1 *(SEREs)*

$r ::= b$		*Boolean expression*
	$[*0]$	*empty SERE*
	$r;\ r$	*sequential concatenation*
	$r\&\&r$	*length – matching conjunction*
	$r[*]$	*repeating r for zero or more times*
	$r[*k]$	*repeating r for k times*
	$r[*n:m]$	*repeating r for n to m times*

In this definition, the *length-matching conjunction* operator && constructs a SERE in which two SEREs both hold at the current cycle, and furthermore both complete in the same cycle.

The set of PSL^{Simple} formulas is defined recursively as follows:

Definition 2.2 *(PSL^{Simple} formulas)*

$f ::= r$		*SERE*
	$b \vee f \mid b \wedge f$	*Boolean operations*
	never r	*negation*
	$X!\ f$	*strong next*
	$X\ f$	*weak next*
	$r \models\!\!\Rightarrow f$	*trigger*
trigger		
	f **until!** $b \mid f$ **until** b	*strong and weak until*
	b **until!**$\_$ $b \mid b$ **until**$\_$ b	*strong and weak overlapping until*(*release*)
	eventually! r	*strong eventually*
	always f	*always*
	f **abort** b	*abnormal termination on b*

In the above definition,

(i) The **until!** constructor is different from **abort**. For f **abort** b, it is unnecessary to verify f any more when b is asserted. But for f **until!** b, even if b is asserted already, one must keep on verifying the strong satisfaction of f on all words starting before the assertion of b.

(ii) **eventually!** r holds in the current cycle of a given path iff the SERE r does hold at the current cycle or at some future cycle during simulation.

Additional PSL^{Simple} temporal operators are treated as syntactic sugars of the above

operators [11].

In dynamic verification, only behaviors with finite length are considered. *PSL* ([3], Section 4.4.5) defines four levels of satisfaction.

(i) **Holds Strongly**, in cases when no bad states have been seen, all future obligations have been met, and the formula will hold on any extension of the word.

(ii) **Holds**, in cases when no bad states have been seen, all future obligations have been met, and the formula may or may not hold on any given extension of the word.

(iii) **Fails**, in cases when a bad state has been seen, future obligations may or may not have been met, and the formula will not hold on any extension of the word.

(iv) **Pending**, in cases when no bad states have been seen, and future obligations have not been met. The formula may or may not hold on any extension of the word.

The precise semantics for PSL^{Simple} is carefully studied in [8] [10] [11] [7] [16].

3 Runs of Alternating Automata: Trees or DAGs

Nowadays, more and more work suggests the AFA as engines for assertion-based verification. AFA wins over Non-Deterministic Büchi Automata in space complexity. The LTL to AFA conversion is linear space [13].

An *alternating finite automaton on finite words* is a tuple of $A =< \Sigma, S, s_0, \rho, F >$, where Σ is the input letter, S is a finite set of states, s_0 is the initial state, $\rho : S \times \Sigma \rightarrow 2^{2^S}$ is a transition function, and F is a finite set of accepting states. The target of a transition is not a state of S, but subsets of S. A state may transit to multiple target sets to express non-deterministic. $\rho(s, l)$ describes all possible configurations of states which A can activate when it is in state s and reads the letter l. For instance, a transition $\rho(s, l) = \{\{s_1, s_2\}, \{s_3, s_4\}\}$ means that A accepts a letter l from state s, and it activates both s_1 and s_2, or both s_3 and s_4.

Traditionally, runs of AFAs are expressed in terms of trees [23] [17]. A finite tree is a finite non-empty set $T \subseteq \mathbb{N}^*$ such that forall $x \cdot c \in T$, with $x \in \mathbb{N}^*$ and $c \in \mathbb{N}$, we have $x \in T$. The elements of T are called nodes, and the empty word ϵ is the root of T. The level of a node x, denoted $| x |$, is its distance from the root ϵ. Particularly, $| \epsilon | = 0$. A run of A on a finite word $w = l_0 \cdot l_1 \cdot \ldots \cdot l_{n-1}$ is a S-labelled tree $< T_r, r >$, where T_r is a tree and $r : T \rightarrow S$ maps each node of T to a state in S. For a $< T_r, r >$, the followings hold:

- $r(\epsilon) = s_0$
- Let $x \in T_r$ with $r(x) = s$ and $\rho(s, l_{|x|}) = \mathbb{S}'$. There is a (possible empty) set $S_K = \{s_1, \ldots, s_k\}$ such that there exists a $S_y \subseteq S_K$ with $S_y \in \mathbb{S}'$, and for all $1 \leq c \leq k$, we have $x \cdot c \in T_r$ and $r(x \cdot c) = s_c$

A run tree r is accepting if all nodes at depth n are labelled by states in F. A word W is accepted iff there is an accepting run on it.

Though AFAs are succinct in expressing LTL formulas, it is difficult to handle

tree-represented AFA at verification time. The difficulties lie in

(i) AFAs do not constrain the breadth of a level. An active state will move to sets of target states whenever values of input variables satisfy corresponding letters. So with the verification process continuing, the memory cost grows without restrictions.

(ii) A tree is just one possible run of an AFA. One have to try breadth-first search or depth-first search in looking for an accepting run.

Kupferman and Vardi [18] [17] proposed to merge *similar* target states of transitions into a single one. That results in representing runs of AFAs by Directed Acyclic Graphs (DAG). For two nodes x_1 and x_2, they are *similar* iff $| x_1 |=| x_2 |$ and $r(x_1) = r(x_2)$. Recently, the DAG approach [14] [5] is accepted in static verification (model checking) of LTL properties. The intuition is that the LTL formulas are equivalent to star-free words. For AFAs converted from LTL formulas, they do not have loops other than self loops. That feature implies that, during verification, one only needs to look in the future, but never the past. Hence, people call runs of traditional LTL-AFAs *memoryless* [17]. In other words, *similar* states correspond to same future mission: to accept the suffixes which satisfy a common property.

Kupferman represents a *memoryless* run $< T_r, r >$ by a DAG $G_r =< V, E >$, where

(i) $V \subseteq S \times \mathbb{N}$ is such that $< s, l > \in V$ iff there exists $x \in T_r$ with $| x |= l$ and $r(x) = s$. For example, $< s_0, 0 >$ is the only vertex of G_r in $S \times \{0\}$.

(ii) $E \subseteq \bigcup_{l \geq 0}(S \times \{l\} \times (S \times \{l+1\}))$ is such that $E(< s, l >, < s', l+1 >)$ iff there exists $x \in T_r$ with $| x |= l$, $r(x) = s$ and $r(x.c) = s'$ for some $c \in \mathbb{N}$.

Configurations $C_i \subseteq S$ are sets of active states, where i refers to the level of a DAG. It is easy to see that, by DAG, every configuration contains at most $| S |$ states that are roots of different subtrees. A DAG is acceptable if there is $C_i \subseteq F$.

One shall note that the branches of AFA's DAGs take resemble the requirements of universal choices. A DAG is just a single path through the existential choices of an AFA. The time-complexity of static LTL verification by AFA is exponential [22] [14]. For dynamic verification, the exponential time-complexity is not released. Feikbeiner and Sipma [9] tried breadth-first, depth-first and backward searchs in checking finite traces using AFA.

4 Local-variable-enhanced Alternating Finite Automata

In [16], we introduced the formalism of Local-variable-enhanced Alternating Finite Automata (LAFA). We use LAFAs to represent PSL^{Simple}formulas.

Definition 4.1 *A LAFA is a tuple of* $A = (V, LV, \Sigma_A, S, s_0, \rho_A, F)$
Where,

- V is the set of variables updated by a DUV.

- LV is the set of local variables of the automaton. V and LV satisfy the following conditions,
  1) $V \cap LV = \phi$
  2) Variables in V do not depend on variables in LV. Updates on LV will not influence variables of V.
- $\Sigma_A = Bool_V \cup FOP_{LV}$ is the letter set of A. We denote by FOP_X the first order predicates over X. We distinguish $\mathbf{true}_V$ and $\mathbf{true}_{LV}$. $\mathbf{true}_V$ stands for logic **true** over V and $\mathbf{true}_{LV}$ stands for logic **true** over LV.
- S is the set of states of an automaton.
- s_0 is the initial state.
- F is the set of states for strong acceptance.
- In LAFA, a transition ρ_A is in type of $S \times \Sigma_A \times U \times 2^S$, where

(i) Σ_A specifies the guarding conditions. A transition can take place only when sampled values of V and LV satisfy its guarding condition. The guarding condition is an expression of either $Bool_V$ or FOP_{LV}. As we verify behaviors of designs against *synchronous* properties, values of variables V are sampled at clock events, such as the occurrences of positive edges and negative edges of clocks. We say a transition is *external* if its guarding condition is the conjunction of a $Bool_V$ expression and a clock expression. Otherwise, we call it an internal transition. The following definition gives the syntax of external conditions.

$$ext\_condition :: Bool_V \wedge posedge(clk) \mid Bool_V \wedge negedge(clk)$$

In the diagrams of our LAFA,
- *Solid arrow*s represent *external* transitions.
- *Dotted arrow*s represent *internal* transitions.
- *Dashed arrow*s represent lines whose types are not important in the context. So a dashed arrow represents either an external or an internal transition.

(ii) U is a set of statements which update local variables whenever the transition takes place. For instance, $\{c_1 := c_1 + 1, c_2 := a\}$ states that c_1 will increase by 1 and c_2 will get the value of a. We do not allow a U to have multiple assignments which updates a common local variable .

(iii) The target of a transition is a subset of S. All elements of the subset shall be active after the transition. Thus, our *LAFA* maintains the universal choice. We realize the existential choices by means of non-deterministic transitions.

(iv) For a state with external transitions, it has an internal transition $(s, \mathbf{true}_{LV}, u_s, \{s\})$. The internal transition is triggered if the state has no more enabled internal transitions and it is still not the time for external transition scheduling. By the internal transition, a state waits in the current state for the next clock event. Thus, samplings on V are executed synchronously.

We adopt DAG in expressing the runs of LAFAs. To avoid search in breadthward, we try to have a DAG to represent all possible runs.

$l \Vdash b$ denotes that the letter l satisfies the Boolean expression b. the Boolean satisfaction relation $\Vdash \subseteq \Sigma \times Bool_V$ behaves in the usual manner.

Definition 4.2 *(Boolean Satisfaction)*

For letter $l \in \Sigma$, *atomic proposition* $p \in P_V$, *and Boolean expressions* $b, b_1, b_2 \in Bool_V$, *then* 1. $l \Vdash p \Leftrightarrow p \in l$

2. $l \Vdash \neg b \Leftrightarrow l \nVdash b$

3. $l \Vdash$ **true** $\wedge\, l \nVdash$ **false**

4. $l \Vdash b_1 \wedge b_2 \Leftrightarrow f \Vdash b_1 \wedge f \Vdash b_2$

Given a predicate g on local variables, $g \in FOP_{LV}$, $C \models g$ denotes the satisfaction of g under the configuration C. That is there exists some states and local variables which make g **true** .

Definition 4.3 *Let* $A = (V, LV, \Sigma_A, S, s_0, \rho_A, F)$ *be a LAFA, runs of* A *over a word* $w = w_0 w_1 w_2 \ldots w_k$ *is a sequence of configurations* $\Delta = C_0 C_1 \ldots C_n$, *where*

(i) $C_0 = \{s_0\}$

(ii) *If* $\exists s_x \in C_i, (s_x, g_x, u_x, S'_x) \in \rho, g_x \in FOL_{LV}$ *and* $g_x \neq$ $\mathbf{true}_{LV}$, *such that* $C_i \models g$, *then for all states* $s \in C_i$ *which has enabled internal transition* (s, g, u, S'), *we have* $S' \subseteq C_{i+1}$

(iii) *If given* w_i, $\exists s \in C_i, (s, g, u, S') \in \rho, g \in ext\_condition$, *and* $w_i \Vdash g$, *then* $S' \subseteq C_{i+1}$

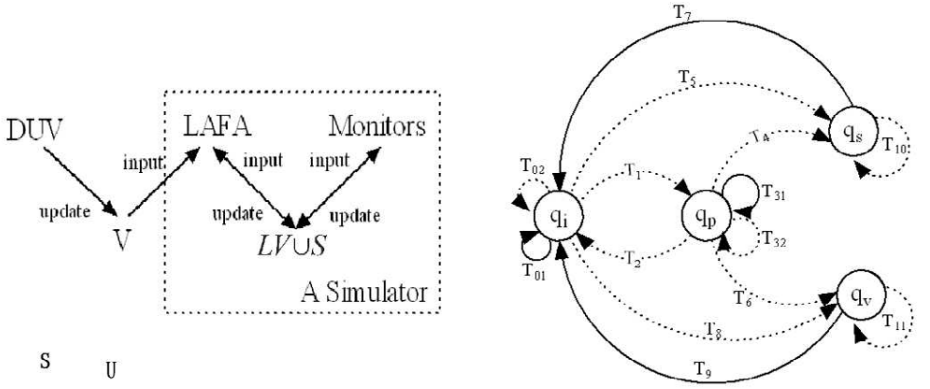
The second clause of Definition 4.3 will not trigger the self-loop transition $(s,$ $\mathbf{true}_{LV}, u_s, S)$ provided that there are active states which have enabled internal guards other than $\mathbf{true}_{LV}$. Meanwhile, the clause removes infinite loops of $\mathbf{true}_{LV}$ guarded transitions which are regarded as *chaos* [15] or *live-lock* [19].

In Definition 4.3, whenever a transition's guarding condition holds, the transition shall take place. That amounts to an identical treatment towards both universal choices and existential choices. However, such a treatment will not impact the correctness of verifying PSL^{Simple} properties. Because in PSL^{Simple}, the acceptance of universal choices asks for synchronization on peer branches. As an example, for $r_1 \&\& r_2$, if the branches of r_1 and r_2 do not synchronize on termination, then the conditions for checking the strong satisfaction of $r_1 \&\& r_2$ will fail. And the strong-accepting instant of f **until**! b is at the moment when f is strongly satisfied by all runs started before the assertion of b. Therefore, we can conclude the violation of a PSL^{Simple} formula only when all its possible runs ended without being accepted.

Owing to the non-determinism in PSL^{Simple} formulas, usually, there is no scheduled synchronization instant. For instance, to $r_1[*3:4]$ $\&\& r_2[*5:6]$, we do not know the exact value of m_1 and m_2, $m_1 \in \{3, 4\}$ and $m_2 \in \{5, 6\}$, such that $r_1[*m_1]$ and $r_2[*m_2]$ are length-matching. So, we must have run-time monitors to

(i) pick up runs which strongly satisfy a PSL^{Simple} formula,

(ii) conclude the strong violation of a PSL^{Simple} formula.



A. Components of our Assertion-Based Dynamic Verifier **B. State Diagram of Monitors**

Fig. 3.

As illustrated in the dotted rectangle of Fig. 3.A, we represent a PSL^{Simple}-Verilog dynamic verifier by

$DV =< A_f, M_o, M_1, \ldots M_f >$, where

- The Verilog DUVs update the variables in V.
- The A_f is the LAFA constructed with respect to formula f. A_f changes its run-time states and local variables in LV on sampling the values of V.
- M_f is the monitors of f. It accept or reject runs A_f.
- M_is are the monitors which watch on the runs of f's sub-formulas. In the bottom-up way, they report and propagate necessary information on which M_f depends in deciding the acceptance and rejection of f.

We define monitors in terms of deterministic finite automata, $M =< \Sigma_M, Q, q_i, \rho_M >$, where

- $\Sigma_M = FOP_{LV \cup S}$ is the letter of M.
- Q is the set of states of a monitor. We denote by
  - q_i the *Idle* state,
  - q_p the *Pending* state, which indicates that the acceptance of a formula is still pending.
  - q_s the *Strong Satisfying* state for indicating the strong satisfaction of a formula.
  - q_v the *Strong Violating* state for indicating the strong violation of a formula.
- Transitions of M are in type of $Q \times \Sigma_M \times U \times Q$, where
  - Σ_M gives out the *guarding conditions*,
  - U is a set of updates on LV and S.

A monitor follows a general behavior template, as shown in **Fig. 3.B**.

- By T_{01} and T_{02}, a monitor stays in the *idle* state q_i provided that the initial state is the only active state in the run time configuration C. T_{01} is an external transition triggered under clock events, T_{02} is the internal one.

- $T_1 =_{df} (q_i, (C \setminus \{s_0\}) \cap S \neq \phi, \phi, q_p)$, which says that if a monitor is in idle state, and the run time configuration contains state other than initial states, then the monitor will move to the *pending* state q_p.
- $T_2 =_{df} (q_p, C = \{s_0\}, \phi, q_i)$ returns a monitor to q_i whenever C contains just initial states again.
- Once the the conditions for strong accepting [16] hold, the transitions T_4 and T_5 move a monitor to the *strong accepting* state q_s. T_4 is triggered from the *pending* state. If the strong accepting holds at the first sampling, T_5 is expected to take place.
- Transitions T_6 and T_8 move a monitor to the *strong violating* state. We will give their definition in the next section.
- By transitions T_{31} and T_{32}, a monitor remains in the *pending state* if none guards of T_4, T_5, T_6 and T_8 holds. T_{31} is triggered on clock events, but T_{32} is an internal transition.
- Once a monitor enters q_s, it will stay in the q_s by the transition T_{10} until the next clock event arrives. And then by the transition T_7, it returns to q_i. The guarding condition of T_7 is $\mathbf{true}_{LV \cup S}$. The situation applies to q_v as well. However, the self-loop transition is T_{11} and returning transition is T_9.

Now, let us have an analysis on the time complexity of the verification process by our dynamic verifiers. Since we have DAGs represent all possible runs, we need not try all branches for searching strong satisfying or violating words. Therefore, the time complexity of our approach is linear to the depth of simulation. The good result comes from features of PSL^{Simple} which emphasizes that

(i) For strong accepting $r_1 \&\& r_2$, words of r_1 and r_2 shall start and stop simultaneously.

(ii) For strong accepting f **until**! b, all words started before the assertion of b shall strongly satisfy f.

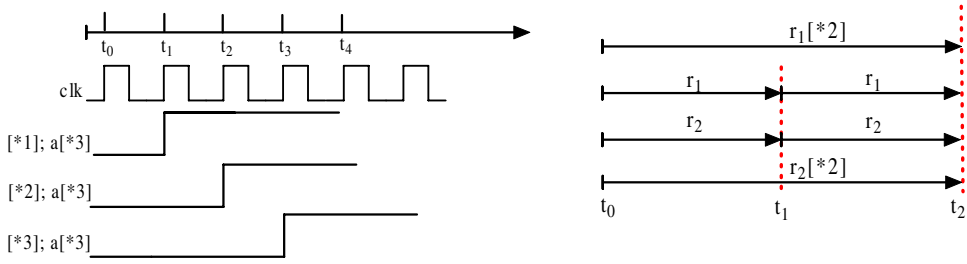
These two conditions amount to require the existence of successful synchronization of all branches for accepting a universal choice. Without such a requirement, we are unable to try all running branches in parallel, and achieve the linear time complexity.

5 Data Types and Encodings of LAFA

In this section, we discuss the run-time techniques for manipulating LAFAs. Firstly, we adopt a symbolic method to encode states. For a state $s \in S$, we use an one-bit variable f_s to flag s's activeness in the current configuration, and f'_s for the next configuration. Given the current configuration C_i, we define

$$f_s \equiv s \in C_i \qquad\qquad f'_s \equiv s \in C_{i+1} \tag{1}$$

If s is in the target state set, command $f'\_s := 1$ activates s in the next configuration. Command $f'\_s := 0$ indicates the deactivation of s. However, such a



A. Words of {[*1:7];a[*3]}

B. PSL^{Simple} formulas are not memoryless

Fig. 4.

command is redundant in our automata. Because, if one does not explicitly specify the command $f'_s := 1$, s will not be active in the next configuration. Actually, we do not allow $f'_s := 0$ to avoid conflicts with other possible $f'_s := 1$.

The symbolic encoding also brings ease to represent values of local variables. Due to the non-determinism of PSL^{Simple}, local variables may have multiple values in a configuration. For example, suppose there is a SERE $\{[*1:7];\ a[*3]\}$ which specifies that after 1 to 7 clock cycles, a shall hold for 3 cycles, then at time t_3, the repetition counter c of a can be 0, 1, and 2, as illustrated in Fig. 4.A. To enable multiple assignments to a local variable, we denote by f_c_e to flag that in the current configuration, the value of variable c equals to e, that

$$f\_c\_e \equiv c == e \tag{2}$$

Same as the approach of state activation, to update a local variable, we only need to assert the new flags. For example, to increase the value of c from 3 to 4, the command is $\{f'\_c\_4 := 1\}$

Recalling that a LAFA transition is a tuple of $\rho = (s, g, U, S')$, where U is the command updating local variables and S' is the target set, we simplify the run-time mechanism on transitions by unifying the operations on states and local variables with the symbolic encoding.

We can not directly apply algorithms for DAGs of LTL-AFAs to ours. Because LTL is a star-free language. But, $PSL^{Simple}$is not star-free. For correct synchronization, branching runs shall remember the time at which they start and fork. Namely, the SERE $\{r_1[*1:2]\ \&\&\ r_2[*1:2]\}[*2:3]$ specifies 2 or 3 times repetition of $\{r_1[*1:2]\&\&r_2[*1:2]\}$. As illustrated in **Fig. 4.B**, when its LAFA manages to synchronize on $r_1[*1]\ \&\&r_2[*1]$ at t_1, it will run for another $r_1[*1:2]\&\&r_2[*1:2]$. Yet, the LAFA may also choose not to synchronize on the first r_1, and continue for $r_1[*2]$. At t_2, when it reaches $r_1[*2]$, it shall synchronize with the $r_2[*2]$ branch started at t_0, not the one forked at t_1.

We accompany each state and local variable with a historical record $h = t_0 t_1 \ldots t_{n-1} t_n$ which logs that at time t_0 the run started, at t_1 there was a universal branch, and the last universal branch took place at t_n. Furthermore, we organize records with the same last branching time into a tree. For instance, in Fig. 5, H_A records three branching runs, one started at t_0, one at t_1 and one at t_5. All three

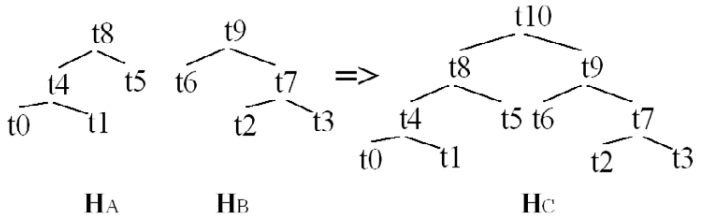


Fig. 5. A Tree of Branching Time

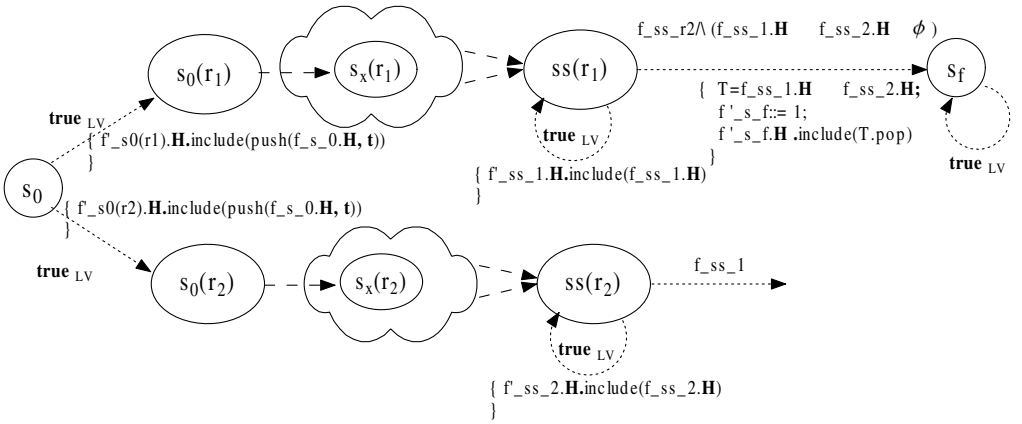


Fig. 6. LAFA of $r_1 \&\& r_2$

runs forked again at t_8.

Given a historical tree H,

(i) function $H.start$ returns all the starting time, they are the leaves of H. For the H_A in Fig. 5, $H_A.start = \{t_0, t_1\}$;

(ii) $H.last$ refers to its root which gives the last branching time;

(iii) $H.pop$ decomposes H by removing its root, and returns a set of sub-trees of H;

(iv) $H.clear(t)$ removes branches which start at t;

(v) Given a set of trees $\mathbf{H}$, $push(\mathbf{H}, t)$ returns a new tree with t as the root and elements of H as subtrees. In Fig. 5, H_C is the result of $push(\{H_A, H_B\}, t_{10})$.

(vi) For a tree set $\mathbf{H}$, $\mathbf{H}.clear()$ empties all its elements. Likewise, $\mathbf{H}.start$, $\mathbf{H}.last$, $\mathbf{H}.pop$, $\mathbf{H}.clear(t)$ work on all elements of $\mathbf{H}$.

$f\_s.\mathbf{H}$ and $f\_c\_v.\mathbf{H}$ denote the historic trees of the state s and the local variable c.

Now, we can give the condition for concluding the strong violation of a property. Let $\mathbf{H}(C)_V$ stand for the historic trees of states which do have enabled out-going transitions in the current configuration C. Let $\mathbf{H}(C)$ stand for the historical trees contained in states of C. Thus $\mathbf{H}(C')$ contains the historic trees of the next configuration.

$g_6 =_{df} \mathbf{H}(C)_V.start \neq \phi \wedge \mathbf{H}(C)_V.start \cap \mathbf{H}(C').start = \phi$

In this definition, $\mathbf{H}(C)_V.start \neq \phi$ says that there are states all whose transitions fail to take place. $\mathbf{H}(C)_V.start \cap \mathbf{H}(C').start = \phi$ says that all triggered transitions do not start from $\mathbf{H}_V$. In other words, all runs starting from $\mathbf{H}_V$ terminate after the current configuration. If g_6 holds, the monitor shall record the starting and ending time of strong violation. That is

$U_6 =_{df} \{q'_v.b := v_{\mathbf{H}}.start,\ q'_v.e := t\}$

The monitor approach releases the obligation of LAFAs to maintain a state for strong violation. That removes a significant amount of LAFA transitions.

With above improvements, we modify the LAFA construction clauses proposed in [16]. Here, we give out the clause for r_1 && r_2, as illustrated in Fig.6

Given $A_i = (V, LV_i, \Sigma, S_i, s0(r_i), \rho_i, \{ss(r_i)\})$ are LAFAs of r_i, then

$$LV(r_1 \&\& r_2) = LV(r_1) \cup LV(r_2)$$

$$S(r_1 \&\& r_2) = S(r_1) \cup S(r_2) \cup \{s_0, s_f\}$$

$$s_0(r_1 \&\& r_2) = s_0$$

$$\begin{aligned}\rho_A(r_1 \&\& r_2) = {} & \rho_A(r_1) \cup \rho_A(r_2) \\ & \cup\{(s_0,\ \mathbf{true}_{LV}, u_i, \{s_0(r_i)\}) \mid u_i = \{l\_s\_0(r_i).\mathbf{H}.include(push(l\_s\_0.\mathbf{H}, t))\}\} \\ & \cup\{(ss(r_1), g, u, \{s_f\}), (ss(r_2), f\_ss\_1, \phi, \phi)\}\end{aligned}$$

$$F(r_1\ \&\&\ r_2) = \{s_f\}$$

where, $g = f\_ss\_r\_2 \wedge (f\_ss\_r\_1.\mathbf{H} \cap f\_ss\_r\_2.\mathbf{H} \neq \phi)$

$$u = \{\ T := f\_ss\_r\_1.\mathbf{H} \cap f\_ss\_r\_2.\mathbf{H};\ f'\_s\_f := 1;\ f'\_s\_f.\mathbf{H}.include(T.pop);\ \}$$

In the above clause, all target states of s_0 inherit s_0's historic trees, and have a new universal branch time as the root of their historic trees. s_f is the strong satisfaction state of $r_1 \&\& r_2$. Before reaching s_f, we shall synchronize on the strong satisfaction of both r_1 and r_2. For the transition from $ss(r_1)$, the guarding condition

$f\_ss\_r\_2 \wedge (f\_ss\_r\_1.\mathbf{H} \cap f\_ss\_r\_2.\mathbf{H} \neq \phi)$

conveys the idea that for a successful synchronization, both $ss(r_1)$ and $ss(r_2)$ shall be active and both of them have common histories of universal choices. The update part activates s_f and assigns the $T.pop$ as histories to s_f. T is a temporary variable. It is just the common histories of $ss(r_1)$ and $ss(r_2)$. $T.pop$ removes the branching time which is pushed into historic tress on leaving s_0. Both $ss(r_1)$ and $ss(r_2)$ are deactivated once the automata reaches s_f.

By this example, we can also see the effect of the $\mathbf{true}_{LV}$ guarded self-loop transition. By the semantics of PSL [7], The length of a SERE is counted on clock events. Internal transitions within two external transitions do not take time. It may takes different internal transitions to reach $ss(r_1)$ and $ss(r_2)$. With the self-loop transition, $ss(r_1)$ will not miss the synchronization with $ss(r_2)$ only if $ss(r_2)$ can be active in the current clock cycle.

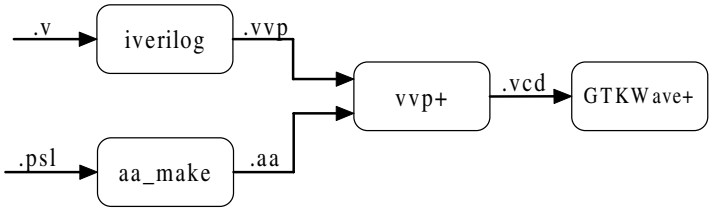


Fig. 7. Architecture of The PSL^{Simple} Dynamic Verifier

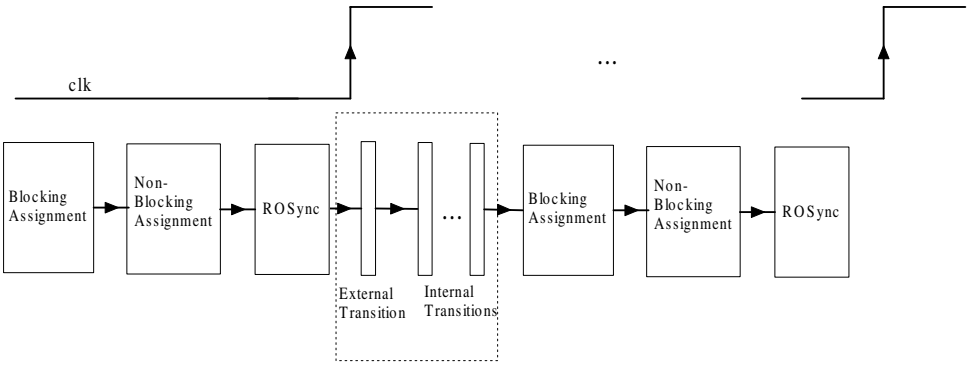


Fig. 8. Sequence of Event Scheduling in VVP+

6 Development of The PSL^{Simple}-Verilog Dynamic Verifiers

We develop the PSL^{Simple}-Verilog Dynamic Verifier in aids of two GPL open source tools, the Icarus-Verilog [24] and the GTKWave [6].

Icarus-Verilog is a package of back-end tool kits for the Verilog HDL as described in the IEEE-1364 standard [1]. It includes a compiler `iverilog`, a simulator `VVP`, an XNF (Xilinx Netlist Format) generator and an EDIF FPGA netlist generator. For batch simulation, the compiler `iverilog` transforms Verilog code into some intermediate assembly codes called vvp. The simulation engine `VVP` reads the vvp assembly codes and outputs the result in Value-Change-Dump (vcd) format. The GTKWave is a wave viewer. It interprets the vcd files and paints the signal values. The Icarus-Verilog/GTKWave combination makes a small but complete Verilog design environment. However, none of them supports assertion-based dynamic verification.

We enhance Icarus-Verilog and GTKWave with new functionalities as plugins. We illustrate the verification flow of PSL^{Simple}-Verilog in Fig.7, where

(i) `aa_make` is a module newly developed by us. It parses and transforms PSL^{Simple} properties to LAFAs. To utilize the existing run-time mechanism of *VVP*, we represent LAFAs in vvp-like assembly codes.

(ii) `VVP+` introduces an extra event schedule phase into `VVP`. The new phase processes LAFA transitions. It is circled in dotted lines of Fig. 8.

The original implementation of `VVP` follows the standard scheduling semantics of Verilog HDL [1]. At each simulation cycle, `VVP` processes in order of

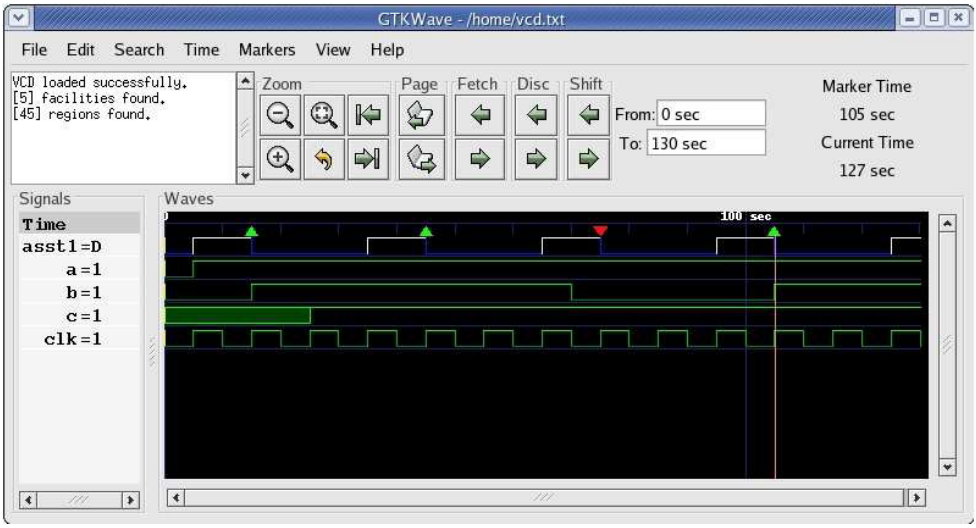


Fig. 9. A Snapshot of the *PSL*^{Simple} Dynamic Verifier

Blocking Assignment events, Non-blocking Assignment events, and the last Read-Only-Sync events. The Blocking Assignment and the Non-Blocking Assignment update the variables in V. The updates may make some Boolean expressions of external transitions hold. For the conjunction of clock expressions in guarding conditions, the external transitions will be postponed until the occurrence of clock events. If some guarding conditions of internal transitions hold after the execution of external transitions, then there will be a sequence of internal transitions. `VVP+` returns to process verilog events when there is no more internal transitions other than the $\mathbf{true}_{LV}$ guarded self-loops.

(iii) `GTKWave+` is developed on the `GTKWave`. `GTKWave+` can graphically express the four states of a property monitor. They are *Idle*, *Pending*, *Strong Satisfying* and *Strong Violating*. When the state of a property turns from *idle* into *pending*, its trace is raised and painted in white. The legend of *Strong Satisfaction* is a upward green triangle and the legend of *Strong Violation* is a downward red triangle. **Fig. 9** is a snapshot of our *PSL*^{Simple}-Verilog dynamic verifier.

7 Discussion and Future Work

In this paper, we have presented methods to make alternating automata from static representation to run-time verification engines.

(i) We have Directed Acyclic Graphs (DAG) represent all possible runs of a LAFA. The acceptance of universal choices depends on successful synchronization of universal branches.

(ii) We encode states and local variables by symbolic approaches, and adopt historic trees in representing all possible parallel runs.

By those methods, we are able to maintain the linear complexity of alternating automata both in space and time.

We also have pointed out that the good result comes from features of PSL^{Simple} which emphasizes that only by successful synchronization of all branches, can we accept universal choices. Without such a requirement, we can not achieve the linear time complexity.

We just finished the prototype of our PSL^{Simple}-Verilog dynamic verifier. In the future, we will have quantitative experiments of our method and propose optimizations.

In addition, we plan to extend our approach to specification assurance methods. Conflicting properties will put verification effort into vain. For developing a specification-centric methodology, we must ensure the consistency of properties before handing them out. Besides these, automatic test generation from $PSL^{Simple}$is also an interesting topic to us.

References

[1] IEEE standard verilog hardware description language, IEEE std 1364-2001.

[2] Y. Abarbanel, I. Beer, L. Gluhovsky, S. Keidar, and Y. Wolfsthal. FoCs: Automatic generation of simulation checkers from formal specifications. *In International Conference on Computer Aided Verification*, LNCS-1855:538–542, 2000.

[3] Accellera. *Property Specification Language Reference Manual Rev 1.1*, 2004.

[4] Shoham Ben-David, Roderick Bloem, Dana Fisman, Andreas Griesmayer, and Ingo Pill Sitvanit Ruah. Automata construction algorithm optimized for psl. Technical Report Delivery 3.2/4, PROSYD, July 2005.

[5] R. Bloem, A. Cimatti, I. Pill, M. Roveri, and S. Semprini. Symbolic implementation of alternating automata. *In Proceedings of The 11th International Conference on Implementation and Application of Automata, LNCS 4094*, pages 208–218, 2006.

[6] Tony Bybell. Gtkwave. http://home.nc.rr.com/gtkwave/index.html.

[7] Cindy Eisner, Dana Fisman, John Havlick, Y Lusting, A McIssac, and D Van Campenhout. Reasoning with temporal logic on truncted paths. *In proceedings of 15th CAV*, LNCS 2725:27–40, 2003.

[8] Cindy Eisner, Dana Fisman, John Havlick, and Johan Mrtensson. The $\top$, $\bot$ appraoch for truncated semantics. Technical report, Accellera, May 2006.

[9] Bernd Feikbeiner and Henny Sipma. Checking finite traces using alternating automata. *Formal Methods in System Design*, Volume 24 ,Issue 2:101–127, 2004.

[10] Dana Fisman. The subset of linear violation. Technical Report MCS05-07, Computer Science and Applied Mathematics, Weizmann Institute of Science, August 2005.

[11] Dana Fisman, Cindy Eisner, and John Havlicek. *Formal syntax and Semantics of PSL: Appendix B of Accellera's Property Specification Language Reference Manual.* Accellera, 1.1 edition, March 2004.

[12] Harry Foster, Adam C. Krolnik, and David J. Lacey. *Assertion-Based Design, Section 1.3.3.* Kluwer Academic Publishers, 2nd edition, 2004.

[13] P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. *In Proceedings of the 13th International Conference on Computer Aided Verification (CAV 2001)*, LNCS-2012:53–65, 2001.

[14] Moritz Hammer. *Linear Weak Alternating Automata and The Model Checking.* PhD thesis, 2005.

[15] C.A.R Hoare and Jifeng He. *Unifying Theories of Programming.* Series in Computer Science. Prentice-Hall Europe, 1st edition, 1998.

[16] Naiyong Jin and Chengjie Shen. Dynamic verifying the properties of the simple subset of psl. *In Proceedings of The first IEEE/IFIP Symposium on Theoretic Aspect on Software Engineering*, pages 229–238, 2007.

[17] Orna Kupferman and Moshe Y. Vardi. Weak alternating automata and tree automata emptiness. *Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing*, pages 224–233, 1998.

[18] Orna Kupferman and Moshe Y. Vardi. Weak alternating automata are not that weak. *ACM Transactions on Computational Logic (TOCL)*, Volume 2, Issue 3:408–429, 2001.

[19] Robin Milner. *Communication and Concurrenvy*. Prentice Hall, 1989.

[20] Dmitry Pidan, Sharon Keidar-Barner, Mark Moulin, and Dana Fisman. Optimized algorithms for dynamic verification. Technical Report Delivery 3.1/1, PROSYD, March 2005.

[21] Daniel Sheridan. Bounded model checking with snf, alternating automata, and büchi automata. *In Proceedings of the 2nd International Workshop on Bounded Model Checking (BMC 2004)*, ENTCS 119,ISSUE 2:83–101, 2004.

[22] Moshe Y. Vardi. Alternating automata and program verification. In *Computer Science Today, LNCS 1000*, pages 471–485. 1995.

[23] Moshe Y. Vardi. An automata-theoretic approach to linaer temporal logic. *Logics for Concurrency : Structure versus Automata*, LNCS-1043:238–266, 1995.

[24] Stephen Williams. Icarus verilog. http://www.icarus.com/eda/verilog/index.html.